Tutorials for queueing simulation

N.D. Van Foreest and E.R. Van Beesten

March 18, 2019

CONTENTS

INTRODUCTION

Typically a queueing system is subject to rules about when to allow jobs to e
adapt the service capacity. Such a decision rule is called a *policy*. The theo
efficacy of policies is often very hard, while with simulation it becomes doal
we present a number of cases to see how simulation can be used to analyze a
systems. Besides the fact that these cases will improve your understanding
probability theory (such as how to compute the empirical cumulative distribut
data analysis, they will also make clear that simulation is a really creative
solving many interesting and challenging algorithmic problems.

Each case is organized in a number of exercises. For each exercise,

1. Make a design of how you want to solve the problem. For instance, make
   system, or a control policy structure, or compute relevant KPIs (key pe
   such as cost, or utilization of the server, and so on). In other words, thi
2. Try to translate your ideas into pseudo code or, better yet, python[1]
3. If you don't succeed in getting your program to work, look up the code w
   it into your python environment.[2].
4. When an exercise has a hint, its marked in the margin with a penguin s
5. Simulate a number of scenarios by varying parameter settings and see

We expect you to work in a groups of 2 to 3 students and bring a lap
and working python environment, preferably the anaconda package avail
`anaconda.com/`, as this contains all functionality we will need[4].

Note that the code is part of the course, hence of the midterms and the exa
as not obligatory, you have to be able to read the code and understand it. Our
or most efficient, rather, we focus on clarity of code so that the underlying re
possible. Once our ideas and code are correct, we can start optimizing, if this

The subsections below provide some extra information regarding the use o
Please read it carefully.

PYTHON BACKGROUND

For the computer simulation assignments in this course it is important t
understanding of the programming language *Python*. Python is arguably e
which you already know, but you have to get used to the syntax. Therefore, i
with Python, we strongly advise you to do the online introductory tutorial on

programming skills anyway (in your studies now but also, with very high pro
professional life). However, for the assignments, the parts outlined below sho

Essential topics in the tutorial:

- INTRODUCTION: completely
- FLOW CONTROL: completely
- FUNCTIONS:
  - Python Function
  - Function argument
  - Python Global, Local and Nonlocal
  - Python Modules
  - Python Package
- DATATYPES: everything except for Python Nested Dictionary
- FILE HANDLING: nothing
- OBJECT & CLASS:
  - Python Class (For assignment 4: *Simulation of the $G/G/1$ queue i*

We will use the following libraries of python a lot:

- `numpy` provides an enormous amount of functions to handle large (mult
  with numbers.
- `scipy` contains numerical recipes, such as solvers for optimization softw
  ential equations. `scipy.stats` contains many probability distributions
  ods to operate on these functions.
- `matplotlib` provides plotting functionality.

We expect you to use google to search for relevant documentation of `numpy` an

A couple of remarks regarding the use of Python on your own laptop:

- Please install Python through the *Anaconda* package (website: `https:/`
  `distribution/#download-section`), since this comes with all the nec
  Select your operating system, click "Download" under "Python 3.7 ve
  steps.

1. Restart the kernel (i.e. the thing/screen in which your output is pr
you can click on a small red cross. In macOS you need to click on th
select “restart kernel”)
2. Remove the line `matplotlib.use('pdf')`, or comment it by puttin
of the line.
3. Now it should work!

1 TUTORIAL 1: EXPONENTIAL DISTRIBUTION

The aim of this tutorial is to show, empirically, a fascinating fact: even for v
in which individuals decide independently to visit a server (a shop, a hospital
distribution is a good model for the inter-arrival times as seen by the serv
simulation to motivate this 'fact of nature'. In particular, our aim is to b
Figures 1, 2 and 3 in terms of cdfs instead of pdfs.

1.1 *Background*

We discuss an example to intuitively see how the exponential distribution o
group of N patients that have to visit a hospital somewhere between 4 and 6 v
up. We assume that we can characterize the inter-arrival times $\{X_k^i, k = 1,2,$
uniform distribution $U[4,6]$ weeks. Then, with $A_0^i = 0$ for all i, define

$$A_k^i = A_{k-1}^i + X_k^i = \sum_{j=1}^{k} X_j^i$$

as the arrival moment of the kth visit of patient i.

Now the hospital doctor 'sees' the superposition of the arrivals of all pati
pute the arrival moments of all patients together is to put all the arrival tim
$1,\ldots,N\}$ into one set, and sort these numbers in increasing order. This result
arrival times $\{A_k, k = 1,2,\ldots\}$ at the doctor of all patients together. Taking A_0

$$X_k = A_k - A_{k-1},$$

is the inter-arrival time between the $k-1$th and kth patient at the doctor. Thu
starting from inter-arrival times of individual patients, we can construct inter
by the doctor.

Suppose that we generate, by means of simulation, many inter-arrival tim
ual patients. Then we compute the arrival times by (1.1), sort these, and c
inter-arrival times $\{X_k\}$ of patients as seen by the doctor. To plot the empirical
of $\{X_k\}$, we just count the number of inter-arrival times smaller than time
empirical distribution of $\{X_k\}$ is defined as

$$\mathsf{P}_n(X \leq t) = \frac{1}{n}\sum_{k=1}^{n} \mathbb{1}_{X_k \leq t},$$

where the *indicator function* is $\mathbb{1}_{X_k \leq t} = 1$ if $X_k \leq t$ and $\mathbb{1}_{X_k \leq t} = 0$ if $X_k > t$.

Let us now compare the probability density as obtained for several simul
density of the exponential distribution, i.e., to $\lambda e^{-\lambda t}$. As a first example, ta

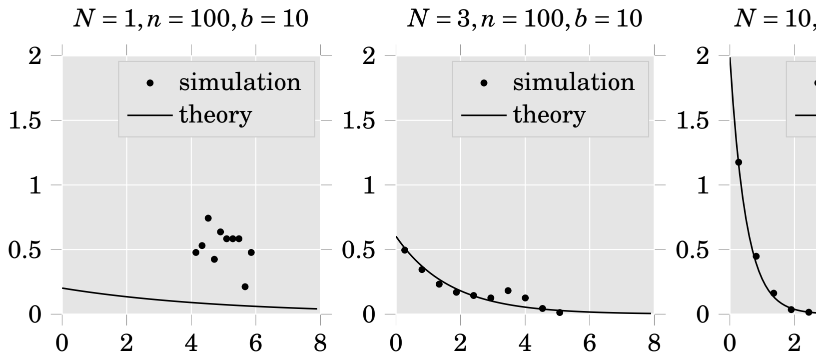

Figure 1: The inter-arrival process as seen by the doctor owner. Observe t
intersects the y-axis at level $N/5$, which is equal to the arrival rate when N p
The parameter $n = 100$ is the simulation length, i.e., the number of visits p
is the number of bins to collect the data. The height of a point correspond
computation is a bit subtle. Let $a = (X_1, \ldots, X_n)$ denote the simulated inter-ar
width of a bin as $\delta = (\max\{a\} - \min\{a\})/b$. Then the height of the ith bin is co
With this: $\sum_{i=1}^{b} h_i \delta = 1$.

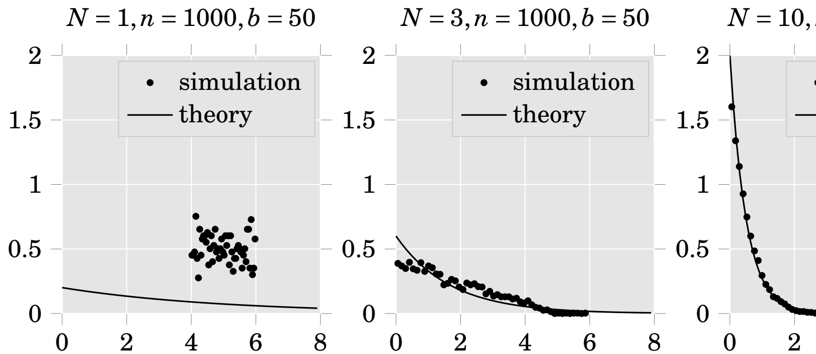

Figure 2: Each of the N patients visits the doctor at uniformly distributed i
now the number of visits is $n = 1000$.

Thus, it appears reasonable to use the exponential distribution to model int
tients for systems (such as a doctor, or a hospital or a call center) that handl
patients each of which deciding independently to visit the system.

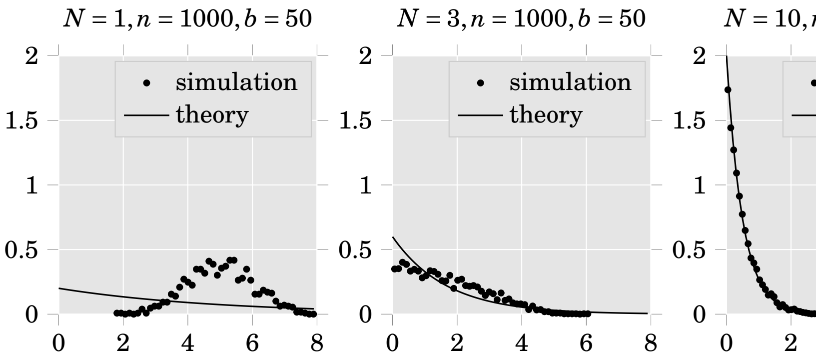

Figure 3: Each of the N patients visits the doctor with normally distributed in
$\mu = 5$ and $\sigma = 1$.

```
import matplotlib
#matplotlib.use('pdf')
import matplotlib.pyplot as plt

# this is to print not too many digits to the screen
np.set_printoptions(precision=3)
```

1.3 *Empirical distributions*

One important step in this process is to compute the empirical distribution.
interesting (and challenging) than you might think[5], we start with this. C
empirical distribution functions, we are in good shape to set up the rest of the

Before designing an algorithm to compute, it is best to start with a simp
and try to formalize the steps we take in the process.

1.2. Suppose you are given the following sample from a population:

```
a = [3.2, 4, 4, 1.3, 8.5, 9]
```

What steps do you take to make the empirical distribution function? Reca

$$F(x) = \frac{\#\{i : a_i \leq x\}}{n},$$

with n is the size of the sample.

Can you turn it into an algorithm? (Just attempt to design an algorith

You should know that for loops in python are quite slow (and for loops i dramatic). For large amounts of data it is better to use `numpy`.

1.7. Use the `numpy` functions `arange`, to replace the `range`, and `sort` to spee the previous exercise.

With the algorithm of Exercise 1.7 we can compute and plot a distribu arrival times specified by a list (vector, array) a. For our present goals this su like details, you should notice that our plot of the distribution function is stil graph should make jumps, but it doesn't. Moreover, our cdf is not a real fun form $x = (1,1,3)$, $y = (0,0.5,1)$. In the rest of this subsection we repair these this if you are not interested.

1.8. Read about the `drawstyle` option of the `plot` function of `matplotlib` jumps.

1.9. Finally, we can make the computation of the cdf significantly faster wit `numpy` functions.

1. `numpy.unique`
2. `numpy.sort`
3. `numpy.cumsum`
4. `numpy.sum`

How can you use these to compute the cdf?

1.4 *Simulating the arrival process of a single patient*

The next step is to simulate inter-arrival times of a single patient and mak these times. Then we graphically compare this cdf to the exponential distrib method to compare cdfs is by means of the Kolmogorov-Smirnov statistic (see develop in passing.

1.10. Generate 3 random numbers uniformly distributed on $[4,6]$. Print these get something decent. Read the documentation of the `uniform` class of in the s Check in particular the `rvs()` function.

1.11. Generate $L = 300$ random numbers $\sim U[4,6]$ and make a histogram o should interpret these random numbers as inter-arrival times of one patient say.)

3. `scipy.stats.uniform`, the `cdf` function.

1.14. Now compute the KS statistics to compare the simulated inter-arrival ti
tial distribution with a suitable mean. (What is this suitable mean?). Do we
that our sample is drawn from this exponential distribution?

See `scipy.stats.expon`.

1.15. Finally, plot the empirical distribution and the exponential distribution
why these graphs are different.

1.5 *Simulating many patients*

We would now like to simulate the inter-arrival process as seen by a doctor
tients. For ease, we call this the merged, or superposed, inter-arrival process
quite a bit of thought. Thus, we start with a numerical example with two pati
steps we make to compute the empirical distribution of the merged inter-arr
make an algorithm, and scale up to many numbers.

1.16. Suppose we have two patients with inter-arrival times $a = [4,3,1.2,5]$ ar
by hand, the empirical cdf of the merged process.

1.17. The steps of the previous exercise can be summarized by the following

```
from itertools import chain

def superposition(a):
    A = np.cumsum(a, axis=1)
    A = list(sorted(chain.from_iterable(A)))
    return np.diff(A)
```

Note that the input `a` in the function `superposition` is a matrix of inter-ar
patients.

Try to understand this code by reading the documentation (on the web)
tions.

1. `numpy.cumsum`, in particular read about the meaning of axis.

2. `itertools.chain.from_iterable`

3. `numpy.diff`

1.18. Generate 100 random inter-arrival times for 3 patients, plot the cdf o

1.6 *Summary*

1.22. Make a summary of what you have learned from this tutorial.

2 TUTORIAL 2: SIMULATION OF THE $G/G/1$ QUEUE IN DISCRETE TI

In this tutorial we simulate the queueing behavior of a supermarket or hospita
service system. We first make a simple model of a queueing system, and the
more and more difficult queueing situations. With these models we can provi
design or improve real-world queueing systems. You will see, hopefully, how a
to evaluate many types of decisions and design problems with simulation.

For ease we consider a queueing system in discrete time, and don't make
the number of jobs in the system and the number of jobs in queue. Thus, que
here to all jobs in the system, cf. Section 1.4 of the queueing book.

2.1 *Set up*

2.1. Write down the recursions to compute the queue length at the end of a
number of arrivals a_i during period i, the queue length Q_0 at the start, and t
s_i. Assume that service is provided at the start of the period. Then sketch an
code) to carry out the computations (use a for loop). Then check this exer
python code.

With the code of the above exercises we can start our experiments.

2.2. Copy the code of the previous exercise to a new file in Anaconda. Then a
run it. Here λ is the arrival rate, μ the service rate, N the number of period
level of the queue. Explain what the code does. Can you also explain the valu
standard deviation?

```
labda, mu, q0, N = 5, 6, 0, 100

a = poisson(labda).rvs(N)
s = poisson(mu).rvs(N)
print(a.mean(), a.std())
```

2.3. Modify the appropriate parts of code of the previous exercise to the belo
what you see.

```
labda, mu, q0, N = 5, 6, 0, 100
a = poisson(labda).rvs(N)
s = poisson(mu).rvs(N)

Q, d = compute_Q_d(a, s, q0)
```

```
Q, d = compute_Q_d(a, s, q0)

plt.plot(Q)
plt.show()
```

Explain what you see.

2.7. Set $q_0 = 10000$ and $N = 1000$. (In Anaconda you can just change the
code again, in other words, you don't have to copy all the code.) Finally, make
times larger.

Explain what you see. What is the drain rate of Q?

2.8. What do you expect to see when $\lambda = 6$ and $\mu = 5$? Once you have formu
check it.

2.2 *What-if analysis*

With simulation we can do all kinds of experiments to see whether the perfo
system improves in the way we want. Here is a simple example of the type c
run now.

For instance, the mean and the sigma of the queue length might be too lar
complain about long waiting times. Suppose we are able, with significant tech
to make the service times more predictable. Then we would like to know the in
on the queue length.

To quantify the effect of regularity of service times we first assume that
exponentially distributed; then we change it to deterministic times. As deter
are the best we can achieve, we cannot do any better than this by just mak
more regular. If we are unhappy about its effect, we have to make service tim
or add extra servers, or block demand so that the inflow reduces.

2.9. Let's test the influence of service time variability. First run this:

```
N = 10000
labda = 5
mu = 6
q0 = 0

a = poisson(labda).rvs(N)
s = poisson(mu).rvs(N)   # marked

Q, d = compute_Q_d(a, s, q0)
```

You should make the crucial observation now that we can experiment wit (system improvements) and compare their effects. In more general terms: si do 'what-if' analyses.

2.3 *Control*

In the previous section we analyzed the effect of the design of the system, average service times. These changes are independent of the queue lengt however, the service rate depends on the dynamics of the queue process. Wh service rates increase; when the queue is small, service rates decrease. This t be seen in supermarkets: extra cashiers will open when the queue length inc

Suppose that normally we have 6 servers, each working at rate 1 per pe becomes longer than 20 we hire two extra servers, and when the queue is emp extra two servers home, until the queue hits 20 again, and so on.

2.11. Try to write pseudo-code (or a python program) to simulate the queue p challenging; it's not a problem if you spend quite some time on this.)

Another way to deal with large queues is to simply block customers if What if we block at a level of 15? How would that affect the average queue a the queue?

2.12. Modify the code to include blocking and do an experiment to see the effe

As a final case consider a single server queue that can be switched on a h associated with keeping a job waiting for one period, there is a cost p to h period, and it costs s to switch on the server. Given the parameter values of t would be a good threshold N such that the server is switched on when the N? We assume (and it is easy to prove) that it is optimal to switch off the ser empty.

2.13. Jobs arrive at rate $\lambda = 0.3$ per period; if the server is present the se period. The number of arrivals and service are Poisson distributed with the gi (without loss of generality), $p = 5$ and $S = 500$. Write a simulator to comput setting $N = 100$. Then, change N to find a better value.

2.14. What have you learned from this tutorial? What interesting extensions can you think of?

2.4 *To be merged*

3 TUTORIAL 3: SIMULATION OF THE $G/G/1$ QUEUE IN CONTINUOU

In this tutorial we will write a simulator for the $G/G/1$ queue. For this we are essential to simulate any stochastic system of reasonable complexity: ar schedule) to keep track of the sequence in which events occur, and *classes* behavior into single logical units. We will work in steps towards our goal; a learn a number of fundamental and highly interesting concepts such as *clas structures*. If you have understood the implementation of the $G/G/1$ queue, dis are no longer a black box for you. Hence, what you will learn from this tutoria the simulation of the $G/G/1$ queueing process.

Once we have built our simulator we apply it to a case in which we analyze at an airport check-in desk with a single server.

In Section 4 we will generalize the simulator to the $G/G/c$ queue and working with a class for the $G/G/c$ queue. We can then also extend the case a

3.1 *Sorting with event stacks*

If we simulate a stochastic system we like to move from event to event. T consider a queueing process such as the $M/M/1$ queue. It is clear that only at epochs something interesting happens; any time in between arrival and de neglected. Hence, in our simulator we prefer not to keep track of the entire ti relevant epochs. If we can do this, we can jump from event to event.

Clearly, we want to follow the sequence of events in the correct order of ti *event stack*. To see how this works, it is best to consider a simple example fir more difficult situations.

Suppose we have 4 students, and we like to sort them in increasing order this is to insert them into a list, but the insertion should respect the orderin very general problem a number of efficient data structures have been develop the *heap queue*.

3.1. Search the web on `python` and `heap queue`. Study the examples and w the students Jan(21), Pete(20), Clair(18), Cynthia(25) in order of increasing a

3.2. Extend the code of the previous exercise such that we can include more ages than just the name, for instance, also the brand of their mobile phone.

3.3. It may seem that we have just solved a simple toy problem, but this is have established something of real importance: heap queues form the core fu all discrete event simulators used around the world. To help you understand to use heap queues to simulate a queueing process.

```
ARRIVAL = 0
DEPARTURE = 1

stack = [] # this is the event stack
```

Note again that we fix the seed so that we get the same random numbers

First of all we need jobs with arrival times and service times. The easiest

a simulator is by means of a class, as in the code below.

```
class Job:
    def __init__(self):
        self.arrival_time = 0
        self.service_time = 0
        self.departure_time = 0
        self.queue_length_at_arrival = 0

    def sojourn_time(self):
        return self.departure_time - self.arrival_time

    def waiting_time(self):
        return self.sojourn_time() - self.service_time

    def __repr__(self):
        return (f"{self.arrival_time}, {self.service_time}, {se
```

A class has a number of *attributes*, such as `self.arrival_time`, to captu
ber of functions, such as `def waiting_time(self)`, to compute specific infor
the class[8]. We initialize the arrival time and service time to zero. Then we sto
and queue length for a statistical analysis at the end of the simulations. Fina
to compute the waiting time and the sojourn time; the `repr` function is used
that our naming of functions also acts as documentation of what the functio
member variables and functions in python start with the word `self`; this is to
of the) member variables from variables with the same name but lying outside

Classes are extremely useful programming concepts, as it enables you t
is, attributes) and behavior (that is, functions that apply to the attributes) in
Moreover, classes can offer functionality to a programmer without the progr
derstand how this functionality is built. Classes offer many more advantage
but we will not discuss that here.

Once we have a class, making an object is simple with this code.[9]

```
while stack:
    time, job, typ = heappop(stack)
    if typ == ARRIVAL:
        handle_arrival(time, job)
    else:
        handle_departure(time, job)
```

3.5. With the above idea to make a while loop, make a list of things that ha arrival event (in particular, what should happen when the server is busy at a should happen if the server is idle?) and at a departure event (in particul is empty, or not empty)? Turn your ideas into code and see whether you ca working. (It's not a problem if you spend some time on this; you will learn a lo

3.3 *Testing*

As a general observation, testing code is very important. For this reason, b simulator, we should apply it to some cases that we can analyze with theory.

3.6. Run a simulation for the $M/M/1$ queue with $\lambda = 2$ and $\mu = 3$ with 100 j average queue length. Then extend to 1000, and then to 10^5. Compare it to th queue length of the $M/M/1$ queue at arrival moments. Do the same thing fo time.

3.7. Thus, let us get further confidence in our $G/G/1$ simulator by specializing Check the documentation of the `uniform` distribution in `scipy.stats` to se code does

```
from scipy.stats import uniform

F = uniform(3, 0.00001)
G = uniform(2, 0.00001)
```

Then use this in our simulation. What happens? What happens if you reve F and G?

3.4 *The check-in process at an airport*

We now have a simulator, and we tested it (although not sufficiently well to cations). We can now start using it. A simple application is to analyze the c airport. For ease we constrain the case here to a single server desk. In the

```
from collections import Counter

c = Counter([j.queue_length_at_arrival for j in served_jobs])
print("Queue length distributon, sloppy output")
print(c)
```

What is your opinion? What would you change to the system?

So, once again, we can use simulation to analyze a practical case and come
ommendations on how to improve the system. It would be interesting to chang
such that the arrival rate becomes time-dependent. That would make the ca
would make the simulator more useful. In fact, the theoretical models nearl
the arrival and service time distribution are constant. When these become d
the way to go. However, let's stop here.

3.5 *Extensions*

Here are some final, more general, questions to deepen your understanding o
simulation.

3.11. Up to now we studied the $G/G/1$ FIFO (First In First Out) queue. How
simulate a LIFO (Last In First Out) queue?

3.12. In a priority queue jobs belong to a priority class. Jobs with higher prio
jobs with lower priority, and within one priority class jobs are served in FIFO
example of a priority queue is the check-in process of business and econom
airports. How would you build a $G/G/1$ priority queue?

3.13. We implemented the queue as a python list. Why is a deque[11] a mor
ture to simulate a queue? Why is it important to know the efficiency of the
algorithms you use?

3.14. Recall that in Exercise 3.4 we built all jobs before the queueing simulati
not a good decision? Note that in the simulation of $G/G/c$ queue below we w
when needed.

3.6 *Summary*

3.15. What have you learned in this tutorial?

4 TUTORIAL 4: $G/G/c$ QUEUE IN CONTINUOUS TIME

The goal of this tutorial is to generalize the simulator for the $G/G/1$ to a G/G
that this is rather easy, once you have the right idea. We will organize the e
single class so that our code is organized and we will develop a method to set
Once we have a working simulator we apply it to the check-in process of the a

4.1 *The simulator*

4.1. In the $G/G/1$ we have just one server which is busy or not. Generalize
function of the $G/G/1$ queue such that it can cope with multiple servers.

4.2. Likewise, generalize the `handle_departure` function of the $G/G/1$ queue
with multiple servers.

4.3. Can you think of some sort of property of the $G/G/c$ queue that must be
important to use such properties as tests while developing.

4.4. The design of the simulation of $G/G/1$ queue is not entirely satisfactor
want to run different experiments, we have to clean up old experiments bef
ones. The current design is not very practical for this. It is better to enca
its behavior (i.e., functions) of the simulator into one class. Build a class tha
queue; use the ideas of the $G/G/1$ simulator.

It is not problem when you spend considerable time on this exercise.
exercises are simple numerical experiments, which should take little time. So
far you get. All code is in the solution.

4.5. How can we instantiate the `GGc` simulator, i.e., make an object of a cla
a simulation with exponential inter-arrival times with $\lambda = 2$, exponential ser
$c = 3$, and 10 jobs?

4.2 *Testing*

4.6. Test the simulator with deterministic inter-arrival times, for instance a
10 minutes, and each job takes 1 hour of service. Check the departure proces
$c = 6$, and $c = 7$.

4.7. For testing purposes, implement Sakasegawa's formula. BTW, why should
formula?

4.8. Test the code for the $M/M/1$ queue with $\lambda = 0.8$ and $\mu = 1$. Compute th

Assume that a flight departing at 11 am consists of $N = 300$ people. For ea
customers arrive between 9 and 10. Suppose that the check-in of a job is $U[1,$

4.11. Model the arrival process.

The queueing problem is evidently to determine the required number of s
performance is acceptable.

4.12. What performance measures are relevant here?

4.13. Define the offered load as $\lambda \mathsf{E}[S]$ and the load as $\rho = \lambda \mathsf{E}[S]/c$. Is it impo
desk that $\rho < 1$?

4.14. Why do we need simulation to analyze this case?

4.15. Search for a good c.

4.16. What do you think of the results? If you are not satisfied, propose an
ment.

4.17. Relate the results of these simulations to your traveling experiences.

4.4 *Summary*

4.18. What have you learned in this tutorial? Can you extend this work to m

5 TUTORIAL 5: AIRPORT CHECK-IN PROCESS

As the simulation of this case is somewhat more difficult than the *G/G/c* queu
important. For this reason we will follow a *test-driven* procedure, which wor
design a number of test cases, from very simple to a bit less simple. Then w
the first test case passes. Then we build and test until also the second case pa
all the tests pass. And, indeed, along the way we caught numerous mista
statements (a very big bug) to simple typos.

5.1 *The simulator*

5.1. Make a list of simple cases whose outcome you can check by hand, to case
but for which you can use theoretical tools to validate the outcome.

Before we build the class for the queues we observe that the implement
Section 4 suffers from a performance penalty. The problem is that we create
objects in the calls `F.rvs()` and `G.rvs()` to generate the quasi-random int
times[13]. Specifically, code like this runs very slow:

```
def generate_jobs_bad_implementation(F, G, p_business, num_jobs
    # the difference in performance is tremendous
    for n in range(num_jobs):
        job = Job()
        job.arrival_time = time + F.rvs()
        job.service_time = G.rvs()
        if uniform(0,1).rvs() < p_business:
            job.customer_type = BUSINESS
        else:
            job.customer_type = ECONOMY

        jobs.add(job)
        time = job.arrival_time

    return jobs
```

Note that we generate `num_jobs` jobs. Then, with probability p (`p_business`
a business customer.

To repair this problem it is better let `scipy` call a random number gener
and generate all random numbers in one pass. Here is one way to obtain a fa
up.

```
            job.customer_type = BUSINESS
        else:
            job.customer_type = ECONOMY

        jobs.add(job)
        time = job.arrival_time

    return jobs
```

We assume that the service distribution is the same for all job classes. I
we can make the service time dependent on the job type.

5.2. How to implement the dependence of job service time on job type?

5.3. Try to implement (or design) the $G/G/c$ queue with business customers. T
this. Business and economy class customers have separate queues and separa
There is one business server and there are c economy class servers. If the busi
becomes idle and there is an economy (business) class customer in queue, th
server takes an economy (business) customer into service.

You can find the code in `tutorial_5.py`. Read it carefully so that you und

5.4. To design some further test cases: think about what happens if two cust
pletely separated: business customers have one dedicated server and the eco
c dedicated servers. How would you implement this in the code?

5.5. Suppose there are no business customers, then all servers should be a
omy class customers. With this idea you can implement the $M/M/c$ queue a
simulator gives the same results as the theoretical values.

5.2 *Case analysis and summary*

Recall, we build this simulator to see whether we want to share the servers or
into the performance of each situation (shared, or unshared), we need to analy
situations. The result of this might be that in certain settings, sharing is
business customer?), and in others, we don't want to share.

5.6. Assume we have $n = 300$ customers, the desks open 2 hours before d
hour before departure, and service distribution is $U[1, 2]$ (minutes) for all cust
business customers is $p = 0.1$. There are $c = 6$ servers for the economy class cu
business customers. Finally, assume that during the opening slot of the desk
is Poisson with constant rate.

HINTS

h.1.16. Note that the doctor sees the combined arrival process, so first fin
arrival times of the patients into one arrival process as observed by the doc
into inter-rival times at the doctor.

h.2.4. For the empirical distribution function you can use the code of Exercise
up, try to recall how the cdf is computed.

h.2.11. As a hint, you need a state variable to track whether the extra serv
The solution shows the code.) Analyze the effect of the threshold at 20; wha
to 18, say, or 30? What is the effect of the number of extra servers; what if yo
of 2?

h.3.6. You might want to check Exercise 3.4 to see how to generate the jobs.

h.3.8. Build a function with arguments λ and G. Use

```
from scipy.stats import uniform
```

to simulate the uniform distribution. (Read the docs to see how to build the u
`scipy.stats`.) Then use `G.var()` and `G.mean()` to compute c_e^2. Then use Litt
expected waiting time to the expected queue length.

h.3.9. Reset to all data and clear all lists.

h.3.12. We have used a good data structure already. How should this be appl

h.3.13. Search the web on `python` and `deque`.

h.4.1. Introduce a `num_busy` variable that keeps track of the number of busy s
if a job arrives and this number is less than c; what if this number is equal to

h.4.2. Again, use the `num_busy` variable.

h.4.3. There must be a relation between the queue length and the number of

h.4.7. Follow the implementation of Exercise 3.8.

h.4.8. Implement the test in a function so that you can organize all your set
ronment. We can use this below for more general cases.

h.4.9. Implement the deterministic distribution as `uniform(1, 0.00001)`.

h.4.11. Most people arrive independently. What is the arrival rate?

SOLUTIONS

s.1.1. 1. Generate realizations of a uniformly distributed random variable r arrival times of one patient.

2. Plot the inter-arrival times.

3. Compute the (empirical) distribution function of the simulated inter-arr

4. Plot the (empirical) distribution function.

5. Generate realizations of uniformly distributed random variables represe times of multiple patients, e.g., 3.

6. Compute the arrival times for each patient.

7. Merge the arrival times for all patients. This is the arrival process as se

8. Compute the inter-arrival times as seen by the doctor.

9. Plot these inter-arrival times.

10. Compare to the exponential distribution function with a suitable arriva

s.1.2. We put the algorithm in a function so that we can use it later. The a study, but it has some weak points. In the exercises below we will repair the p

```
def cdf(a):
    a = sorted(a)
    m, M = int(min(a)), int(max(a))+1
    # Since we know that a is sorted, this next line
    # would be better, but less clear perhaps:
    # m, M = int(a[0]), int(a[-1])+1

    F = dict() # store the function i \to F[i]
    F[m-1]=0  # since F[x] = 0 for all x < m
    i = 0
    for x in range(m, M):
        F[x] = F[x-1]
        while i< len(a) and a[i] <= x:
            F[x] += 1
            i += 1
```

```
I = range(0, len(a))
s = sorted(a)
plt.plot(I, s)
plt.show()
```

s.1.5. Here is one way. Note that we already imported `matplotlib`, so we don

```
def cdf(a):
    y = range(1, len(a)+1)
    y = [yy/len(a) for yy in y] # normalize
    x = sorted(a)
    return x, y

x, y = cdf(a)

plt.plot(x, y)
plt.show()
```

s.1.6. The reason is that at s_1 the first observation occurs. Hence, F should m one at s_1. Next, the `range` function works up to, but not including, its seco (in code), `range(10)[-1]/10 = 0.9`, that is, the last element `range(10)[-]` $0,1,\dots,9$ is not 10. Hence, when we extend the range to `len(a)+1` we ha including the element we want to include.

s.1.7. This code is much, much faster, and also very clean. Note that we norm

```
def cdf(a):
    y = np.arange(1, len(a)+1)/len(a)
    x = np.sort(a)
    return x, y
```

s.1.8. With the `drawstyle` option:

```
plt.plot(x, y,  drawstyle = 'steps-post')
plt.show()
```

But now we still have vertical lines. To remove those, we can use `hlines`.

```
y = range(0, len(a)+1)
y = [yy/len(a) for yy in y] # normalize
s = sorted(a)
left = [min(s)-1] + s
```

```
        #make a plot like before
        yy = [0] + list(y)
        left = [min(x)-1] + list(x)
        right = list(x) + [max(x) + 1]

        plt.hlines(yy, left, right)
        plt.show()

    return x, y


cdf(a, True)
```

s.1.10. Copy this code and run it.

```
from scipy.stats import uniform

# fix the seed
scipy.random.seed(3)

# parameters
L = 3  # number of inter-arrival times

G = uniform(loc=4, scale=2) # G is called a frozen distribution
a = G.rvs(L)
print(a)
```

s.1.11. Add this code to the other code and run it.

```
N = 1 # number of patients
L = 300 # number of interarrival times
a = G.rvs(L)

plt.hist(a, bins=int(L/20), label="a") #bins of size 20
plt.title("N = {}, L = {}".format(N, L))
plt.legend()
plt.show()
```

s.1.12. Add this to the other code and run it.

```
x, y = cdf(a)
```

s.1.14. Add this to the other code and run it. Since the mean inter-arrival tim

```
from scipy.stats import expon

labda = 1./5 # lambda is a function in python
E = expon(scale=1./labda)
print(E.mean()) # to check that we chose the right scale
print(KS(a, E))
```

s.1.15. Add this to the other code and run it.

```
x, y = cdf(a)
dist_name = "U[4,6]"
def plot_distributions(x, y, N, L, dist_name):
    # plot the empirical cdf and the theoretical cdf in one fig
    plt.title("X ~ {} with N = {}, L = {}".format(dist_name,N,
    plt.plot(x, y, label="empirical")
    plt.plot(x, E.cdf(x), label="exponential")
    plt.legend()
    plt.show()

plot_distributions(x, y, N, L, dist_name)
```

It is pretty obvious why these graphs must be different: we compare a uni
tial distribution.

s.1.16. The following steps in code explain the logic.

```
a=[4, 3, 1.2, 5]
b=[2, 0.5, 9]

def compute_arrivaltimes(a):
    A=[0]
    i = 1
    for x in a:
        A.append(A[i-1] + x)
        i += 1

    return A

A = compute_arrivaltimes(a)
```

```
E = expon(scale=1./(N*labda))
print(E.mean())

x, y = cdf(a)
dist_name ="U[4,6]"
plot_distributions(x, y, N, L, dist_name)

print(KS(a, E)) # Compute KS statistic using the function defin
```

s.1.19. Add this to the other code and run it.

```
N, L = 10, 100
a = superposition(G.rvs((N, L)))

E = expon(scale=1./(N*labda))

x, y = cdf(a)
dist_name ="U[4,6]"
plot_distributions(x, y, N, L, dist_name)

print(KS(a, E))
```

This is great. For just $N = 10$ we see that the exponential distribution is a

s.1.20. Add this to the other code and run it.

```
from scipy.stats import norm

N, L = 10, 100

N_dist = norm(loc=5, scale=1)
a = superposition(N_dist.rvs((N, L)))
x, y = cdf(a)
dist_name = "N(5,1)"
plot_distributions(x, y, N, L, dist_name)

print(KS(a, E))
```

Clearly, whether the distribution of inter-arrival times of an individual p normal, doesn't really matter. In both cases the exponential distribution is a go

5. This convergence is not so sensitive to the distribution of the inter-arr patients. For the doctor, only the population matters.

6. Using functions (e.g., `def compute(a)`) to document code (by the func plexity, and reuse code so that it can be applied multiple times. Moreov is in line with the extremely important Don't-Repeat-Yourself (DRY) pr

7. Coding skills: python, `numpy` and `scipy`.

s.2.1. We need a few imports.

```
import numpy as np
import scipy
from scipy.stats import poisson
import matplotlib.pyplot as plt

scipy.random.seed(3)


def compute_Q_d(a, s, q0=0):
    d = np.zeros_like(a)
    Q = np.zeros_like(a)
    Q[0] = q0 # starting level of the queue
    for i in range(1, len(a)):
        d[i] = min(Q[i-1], s[i])
        Q[i] = Q[i-1] + a[i] - d[i]

    return Q, d
```

One of the nicest things about python is that the real code and pseudo cod so much.

s.2.2. Here is the complete code in case you have messed things up.

```
import numpy as np
import scipy
from scipy.stats import poisson
import matplotlib.pyplot as plt

scipy.random.seed(3)
```

s.2.3. You should see that the queue starts at 100 and drains roughly at ra
Q_0 very large (10000 or so), you should see that the queue length process beh
until it hits 0.

s.2.4. This it the code.

```
print(d.mean())

x, F = cdf(Q)
plt.plot(x, F)
plt.show()
```

s.2.5. The mean number of departures must be (about) equal to the mean
period. Jobs cannot enter from 'nowhere'.

s.2.6. The queue length drains at rate $\mu - \lambda$ when q_0 is really large. Thus,
might just as well approximate the queue length behavior as $q(t) = q_0 - (\mu - \lambda$
istic system.

s.2.7. You should see that the queue drains with relatively less variations
'straighter'.

s.2.8. The queue should increase with rate $\lambda - \mu$. Of course the queue length
a bit around the line with slope $\lambda - \mu$, but the line shows the trend.

s.2.9. You should observe that the mean and the variability of the queue leng

s.2.10. For our experiments it is about 20% extra.

s.2.11. Here is one way.

```
def compute_Q_d(a, q0=0, mu=6, threshold=20, extra=2):
    d = np.zeros_like(a)
    Q = np.zeros_like(a)
    Q[0] = q0
    present = False # extra employees are not in
    for i in range(1, len(a)):
        rate = mu + extra if present else mu # service rate
        s = poisson(rate).rvs()
        d[i] = min(Q[i-1],s)
        Q[i] = Q[i-1] + a[i] - d[i]
```

```
def compute_Q_d(a, s, q0=0, b=np.inf):
    # b is the blocking level.
    d = np.zeros_like(a)
    Q = np.zeros_like(a)
    Q[0] = q0
    for i in range(1, len(a)):
        d[i] = min(Q[i-1], s[i])
        Q[i] = min(b, Q[i-1] + a[i] - d[i])

    return Q, d


N = 10000
labda = 5
mu = 6
q0 = 0

a = poisson(labda).rvs(N)
s = poisson(mu).rvs(N)

Q, d = compute_Q_d(a, s, q0, b=15)
print(Q.mean(), Q.std())

x, F = cdf(Q)
plt.plot(x, F)
plt.show()
```

s.2.13. Here is all the code.

```
num_jobs = 10000
labda = 0.3
mu = 1
q0 = 0
N = 100 # threshold

h = 1
p = 5
S = 500
```

```
        Q[i] = Q[i-1] + a[i] - d[i]
        if Q[i] == 0:
            present = False  # send employee home
        elif Q[i] >= N:
            if present == False:
                present = True # server is switched on
                setup_cost += S
        queueing_cost += h * Q[i]

    print(queueing_cost, setup_cost, server_cost)

    total_cost = queueing_cost + server_cost + setup_cost
    num_periods = len(a) - 1
    average_cost = total_cost / num_periods
    return average_cost

a = poisson(labda).rvs(num_jobs)
av = compute_cost(a, q0)
print(av)
```

After a bit of experimentation, we see that $N = 15$ is quite a bit better than N

s.2.14. Some important points are the following.

1. Making a simulation requires some ingenuity, but is often not difficult.
2. With simulation it becomes possible to analyze many difficult queueing ematical analysis is often much harder, if possible at all.
3. We studied the behavior of queues under certain control policies, typical the service rate as a function of the queue length. Such policies are use in supermarkets, hospitals, the customs services at airports, and so on. some tools to design and analyze such systems.

An interesting extension is to incorporate time-varying demand. In m supermarkets, the demand is not constant over the day. In such cases th should take this into account.

s.2.15. Below I fix the seed of the random number generator to ensure that results from the simulator. The arrival process is Poisson, and the number 21 per period. I use `Q = np.zeros_like(a)` to make an array of the same

```
>>> c = mu * np.ones_like(a)
>>> c
array([21, 21, 21, 21, 21, 21, 21, 21, 21, 21])
```

Now for the queueing recursions:

```
>>> Q = np.zeros_like(a)
>>> d = np.zeros_like(a)
>>> Q[0] = 10  # initial queue length

>>> for k in range(1, len(a)):
...     d[k] = min(Q[k - 1], c[k])
...     Q[k] = Q[k - 1] - d[k] + a[k]
...
```

Here are the departures and queue lengths for each period:

```
>>> d
array([ 0, 10, 17, 14, 10, 21, 21, 19, 17, 19])
>>> Q
array([10, 17, 14, 10, 22, 23, 19, 17, 19, 21])
```

Suppose we define loss as the number of periods in which the queue le
course, any other threshold can be taken. Counting the number of such p
Python: (Q>20) gives all entries of Q such $Q > 20$, the function sum() just ad

```
>>> loss = (Q > 20)
>>> loss
array([False, False, False, False,  True,  True, False, False,
        True])
>>> loss.sum()
3
```

Now all statistics:

```
>>> d.mean()
14.8
>>> Q.mean()
17.2
>>> Q.std()
4.377213725647858
>>> (Q > 20).sum()
```

```
...     Q[k] = Q[k - 1] - d[k] + a[k]
...
>>> d.mean()
20.178
>>> Q.mean()
28.42
>>> Q.std()
10.155865300406461
>>> (Q > 30).sum()/num * 100
34.2
```

I multiply with 100 to get a percentage. Clearly, many jobs see a long queu some 28 jobs. For this arrival rate a service capacity of $\mu = 21$ is certainly too

Hopefully you understand from this discussion that once you have the recu the queueing process, you are 'in business'. The rest is easy: make plots, do s semble statistics, vary parameters for sensitivity analysis, and so on.

s.2.16. In this example the number of jobs served per day is $\sim P(21)$, so the but the period capacity c is a Poisson random variable.

```
>>> c = np.random.poisson(mu, num)
>>> Q = np.zeros_like(a)
>>> d = np.zeros_like(a)
>>> Q[0] = 10  # initial queue length

>>> for k in range(1, len(a)):
...     d[k] = min(Q[k - 1], c[k])
...     Q[k] = Q[k - 1] - d[k] + a[k]
...
>>> d.mean()
20.148
>>> Q.mean()
42.518
>>> Q.std()
22.841096208369684
>>> (Q > 30).sum()/num * 100
62.4
```

Comparing this to the result of the previous exercise in which the service c was constant $c = 21$, we see that the average waiting time increases, just as t

```
while stack:
    age, name = heappop(stack)
    print(name, age)
```

Pushing puts things on the stack in a sorted fashion, popping takes things
we put the students on the stack. To print, we remove items from the stack u

s.3.2. We can make the tuples, i.e., the data between the brackets, just longer

```
from heapq import heappop, heappush

stack = []

heappush(stack, (21, "Jan", "Huawei"))
heappush(stack, (20, "Piet", "Apple"))
heappush(stack, (18, "Klara", "Motorola"))
heappush(stack, (25, "Cynthia", "Nexus"))

while stack:
    age, name, phone = heappop(stack)
    print(age, name, phone)
```

s.3.3. For a queueing process we can start with putting a number of job arri
label these events as 'arrivals'. Whenever a service starts, compute the depar
put this departure moment on the stack. Label this event as a 'departure'. T
event. Since the event stack is sorted in time, the event at the head of the sta
in time something useful happens.

More generally, a discrete-time stochastic process moves from event to e
tain actions have to be taken, and these actions may involve the generation
removal of old events. The new events are put on the stack, the obsolete on
the simulator moves to the first event on the stack.

s.3.4. One way is like this.

```
labda = 2.
mu = 3.
rho = labda/mu
F = expon(scale=1./labda)  # interarrival time distributon
G = expon(scale=1./mu)  # service time distributon

num_jobs = 10
```

```
class Server:
    def __init__(self):
        self.busy = False

server = Server()
queue = []
served_jobs = [] # used for statistics

def start_service(time, job):
    server.busy = True
    job.departure_time = time + job.service_time
    heappush(stack, (job.departure_time, job, DEPARTURE))

def handle_arrival(time, job):
    job.queue_length_at_arrival = len(queue)
    if server.busy:
        queue.append(job)
    else:
        start_service(time, job)

def handle_departure(time, job):
    server.busy = False
    if queue: # queue is not empty
        next_job = queue.pop(0) # pop oldest job in queue
        start_service(time, next_job)

while stack:
    time, job, typ = heappop(stack)
    if typ == ARRIVAL:
        handle_arrival(time, job)
    else:
        handle_departure(time, job)
        served_jobs.append(job)
```

s.3.6. Now we see that we needed to store the jobs in `served_jobs`. Set

```
num_jobs=100
```

in the code of Exercise 3.4. Put the following code at the end of the simu
statistics.

s.3.7. With `F=uniform(3, 0.00001)` the job inter-arrival times are nearly 3. times are nearly 2. Hence, arriving customers will always see an empty que average queue length upon arrival equals zero. When you reverse the 2 and 3 up.

s.3.8. First we compute the average waiting time. Then we use Little's law t time to the expected queue length.

```
def pollaczek_khinchine(labda, G):
    ES = G.mean()
    rho = labda*ES
    ce2 = G.var()/ES/ES
    EW = (1.+ce2)/2 * rho/(1-rho)*ES
    return EW


labda = 1./3
F = expon(scale=1./labda)  # interarrival time distributon
G = uniform(1, 2)

print("PK: ", labda*pollaczek_khinchine(labda,G))
```

s.3.9. To ensure that we do not keep old data, we have to reset all lists.

```
stack = [] # this is the event stack
queue = []
served_jobs = [] # used for statistics

job = Job()

num_jobs = 100000

time = 0
for i in range(num_jobs):
    time += F.rvs()
    job = Job()
    job.arrival_time = time
    job.service_time = G.rvs()
    heappush(stack, (job.arrival_time, job, ARRIVAL))
```

```
print("Theo avg. sojourn time: ", pollaczek_khinchine(labda,G)
print("Simu avg. sojourn time:", av_sojourn_time)
```

s.3.10. I ran the code for $n = 100000$ and got this

```
Counter({0: 37134, 1: 16558, 2: 12517, 3: 9070, 4: 6725, 5: 478

   8: 2047, 9: 1433, 10: 1095,  11: 818, 12: 545, 13: 401, 14:

   15: 178, 16: 116, 17: 72, 18: 49, 19: 27, 20: 13, 21: 6, 22:
```

So, about 2% see a queue that is longer than 10, and about 10% (which balling) of the people see a queue that is longer than 5. It seems that the s small, assuming that the customers have a flight to catch.

s.3.11. This is really easy: change the line `queue.pop(0)` by `queue.pop()`.

s.3.12. As in the LIFO example, we only have to change the data structure job an extra attribute corresponding to its priority. Then we use a heap to st Specifically, change the line with `queue.append(job)` by

```
heappush(queue, (job.priority, job))
```

and `queue.pop(0)` by `heappop(queue)`.

s.3.13. In a python list the `pop(0)` function is an $O(n)$ operation, where n is th in the list. In a deque appending and removing items to either end of the dequ

When you do large scale simulations, involving many hours of simulation build up and can make the running time orders of magnitude longer. A fa sorting of numbers. A simple, but stupid, sorting algorithm is $O(n^2)$ whil $O(n \log n)$. The sorting time of 10^6 numbers is dramatically different.

For our code, add the line

```
from collections import deque
```

and replace `queue = []` by

```
queue = deque()
```

and `queue.pop(0)` by

```
queue.popleft()
```

s.4.2. At a departure, a server becomes free, hence decrease `num_busy` by on
queue, start a new service.

s.4.3. Of course, the number of busy servers cannot be negative or larger tha
cannot be negative. Finally, it cannot be that the queue length is positive an
servers is less than c.

s.4.4. See `tutorial_4.py`. Study it in detail so that you really understand h

s.4.5. First we instantiate, then we run and print.

```
labda = 2
mu = 1
ggc = GGc(expon(scale=1./labda), expon(scale=1./mu), 3, 10)
ggc.run()

print(ggc.served_jobs)
```

s.4.7. Recall that Sakasegawa's formula is exact for the $M/G/1$ queue, henc
queue.

```
def sakasegawa(F, G, c):
    labda = 1./F.mean()
    ES = G.mean()
    rho = labda*ES/c
    EWQ_1 = rho**(np.sqrt(2*(c+1)) - 1)/(c*(1-rho))*ES
    ca2 = F.var()*labda*labda
    ce2 = G.var()/ES/ES
    return (ca2+ce2)/2 * EWQ_1
```

s.4.8. In the next function we perform the test. We give this function standa
we can run the function as a standalone test. As such we want to to contain a

```
def mm1_test(labda=0.8, mu=1, num_jobs=100):
    c = 1
    F = expon(scale=1./labda)
    G = expon(scale=1./mu)

    ggc = GGc(F, G, c, num_jobs)
    ggc.run()
    tot_wait_in_q = sum(j.waiting_time() for j in ggc.served_jo
```

```
    ggc = GGc(F, G, c, num_jobs)
    ggc.run()
    tot_wait_in_q = sum(j.waiting_time() for j in ggc.served_jo
    avg_wait_in_q = tot_wait_in_q/len(ggc.served_jobs)

    print("M/D/1 TEST")
    print("Theo avg. waiting time in queue:", sakasegawa(F, G,
    print("Simu avg. waiting time in queue:", avg_wait_in_q)

md1_test(num_jobs=100)
md1_test(num_jobs=10000)
```

s.4.10. The code.

```
def md2_test(labda=1.8, mu=1, num_jobs=100):
    c = 2
    F = expon(scale=1./labda)
    G = uniform(mu, 0.0001)

    ggc = GGc(F, G, c, num_jobs)
    ggc.run()
    tot_wait_in_q = sum(j.waiting_time() for j in ggc.served_jo
    avg_wait_in_q = tot_wait_in_q/len(ggc.served_jobs)

    print("M/D/2 TEST")
    print("Theo avg. waiting time in queue:", sakasegawa(F, G,
    print("Simu avg. waiting time in queue:", avg_wait_in_q)

md2_test(num_jobs=100)
md2_test(num_jobs=10000)
```

s.4.11. Since most people arrive independent of each other, it is reasonable th
in this period of duration T is Poisson with rate $\lambda = N/T$.

s.4.12. It is reasonable *within the context* that the mean waiting time is les
largest waiting time and the fraction of people that wait longer than 10 mi
context is crucial when making models.

s.4.13. There are, by assumption, only arrivals between 9 and 10, hence the
stationary, hence the queueing process is not stationary. It is also not a pro

```
    longer_ten = sum((j.waiting_time()>=10)for j in ggc.served_
    return max_waiting_time, longer_ten


def intake_test_1():
    labda = 300/60
    F = expon(scale=1.0 / labda)
    G = uniform(1, 2)
    num = 300

    print("Num servers, max waiting time, num longer than 10")
    for c in range(3, 10):
        max_waiting_time, longer_ten = intake_process(F, G, c,
        print(c, max_waiting_time, longer_ten)


intake_test_1()
```

s.4.16. I get these numbers

```
Num servers, max waiting time, num longer than 10
3 138.1697966325396 280
4 82.7792212258584 266
5 54.741743193174834 227
6 39.35432320034931 220
7 22.868514027211248 167
8 15.77953068102207 102
9 12.118348481100451 50
```

Even for 9 servers, 50 people wait longer than 10 minutes. Interesting
maximum waiting time is just 12 minutes, so we can at least keep this withir

We could also make a plot of the job waiting time as a function of time, bu

Perhaps we should open the check-in desks for a longer time and advise
Let's try 90 minutes instead.

```
def intake_test_2():
    labda = 300/90
    F = expon(scale=1.0 / labda)
    G = uniform(1, 2)
    num = 300
```

3. Organizing cases and experiments by means of functions. With these fu
log of what we precisely tested, what parameter values we used and whi
tests are also useful to show how to actually use/run the code.

Some interesting extensions.

1. Scale up to networks of $G/G/c$ queues.

2. In the $G/G/c$ queue the assumption is that all servers have the same se
tion settings this is typically not the case. There is a queue of jobs, and
by machines with different speeds. For instance, old machines may work
new machines.

s.5.1. Here are two tests. The tests will not yet work since we have not y
simulate the queueing system. In other words, we know that our first test sho

```python
def DD1_test_1():
    # test with only business customers
    c = 0
    F = uniform(1, 0.0001)
    G = expon(0.5, 0.0001)
    p_business = 1
    num_jobs = 5
    jobs = generate_jobs(F, G, p_business, num_jobs)
    ggc = GGc_with_business(c, jobs)
    ggc.run()
    ggc.print_served_job()


def DD1_test_2():
    # test with only economy customers
    c = 1
    F = uniform(1, 0.0001)
    G = expon(0.5, 0.0001)
    p_business = 0
    num_jobs = 5
    jobs = generate_jobs(F, G, p_business, num_jobs)
    ggc = GGc_with_business(c, jobs)
    ggc.run()
    ggc.print_served_job()
```

```
            job.service_time = Gb.rvs()
        else:
            job.customer_type = ECONOMY
            job.service_time = Ge.rvs()

        jobs.add(job)
        time = job.arrival_time

    return jobs
```

s.5.8. 1. First, check whether the our simulator is indeed correct. Is ther business class customers? If so, do the business server(s), when idle, class customers in service?

2. You'll need real data to see whether our simple arrival process and ser realistic. I suppose most of this is already known. For instance, the nu the plain is known, as is the number of business customers. Service tim the desks, either by measuring, or by using the times the boarding pass

3. Once you have good data, vary one parameter at a time, and report th People typically want to see trends; graphs are indispensable for this.

Finally, you often need to compare a host of different policies:

- Perhaps it is better to have dynamic service capacity, 3 during the first
- Perhaps the service distributions of the business customers can be made regular. What would be impact of this?
- Since there is small number of business customers, why not open the b later than the other desks? Or, share the desks during the first opening 'unshare' during the second?